北京邮电大学课程设计报告

<table>
<tr><td>课程设计名称</td><td colspan="2">计算机网络
课程设计</td><td>学 院</td><td>计算机</td><td>指导教师</td><td>吴起凡</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>318</td><td>17</td><td colspan="2">2018211125</td><td>路已人</td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td>课程设计内容</td><td colspan="6">设计一个 DNS 服务器程序，读入“域名-IP 地址”对照表，当客户端查询域名对应的 IP 地址时，用域名检索该对照表，得到三种检索结果：
1. 检索结果为 ip 地址 0.0.0.0，则向客户端返回“域名不存在”的报错消息（不良网站拦截功能）
2. 检索结果为普通 IP 地址，则向客户返回这个地址（服务器功能）
3. 表中未检到该域名，则向因特网 DNS 服务器发出查询，并将结果返给客户端（中继功能）
课程设计的程序实现了对多个客户端提供服务的功能，并且在本地设置缓存区以提高服务效率。缓存区的替换方式为 LRU，同时使用优先队列来进行过期数据的删除。
程序使用 C 语言完成，使用 C 语言的 socket API 来进行网络通信。</td></tr>
<tr><td>学生课程设计报告
(附页)</td><td colspan="6"></td></tr>
<tr><td>课程设计成绩评定</td><td colspan="6">评语：

成绩：

指导教师签名：
年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

1 DNS 协议简介

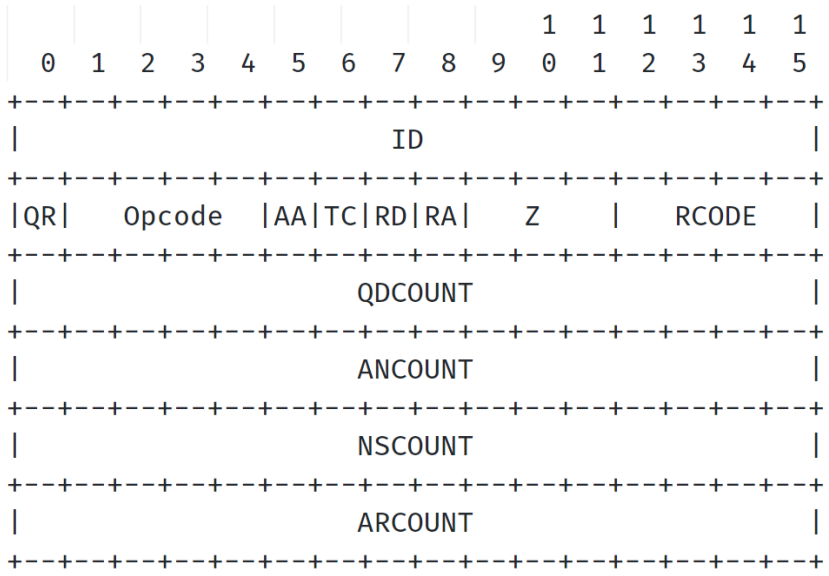
域名系统（Domain Name System, DNS）是互联网的一项服务，它将域名和 IP 地址进行相互映射。为了进行完备的域名-IP 互相转换，DNS 协议包含了多种资源记录类型，例如：

- A 记录：用于名称解析的重要记录，它将特定的主机名映射到对应主机的 IP 地址上。
- CNAME：用于将某个别名指向到某个 A 记录上。
- AAAA 记录：用于将特定的主机名映射到一个主机的 IPv6 地址。

DNS 的协议在应用层上进行通信，使用 TCP 或 UDP 的 53 端口发送数据报文，报文由 Header、Question、Answer、Authority、Additional 五个部分组成。

1.1 Header

为报文的首部，格式如下：

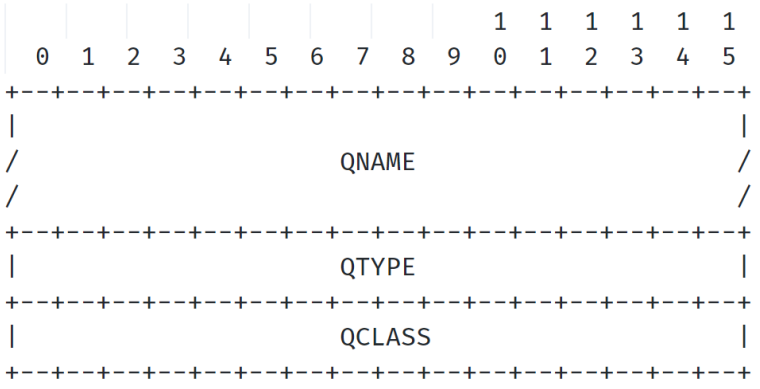


- ID：由客户程序设置，发送报文的编号。
- QR：0 表示查询报，1 表示响应报。
- OPCODE：通常值为 0（标准查询），其他值为 1（反向查询）和 2（服务器状态请求）。
- AA: 权威答案(Authoritative answer)。
- TC: 截断的(Truncated)，表示应答的总长度超 512 字节，此时只返回前 512 个字节。
- RD: 期望递归(Recursion desired)，请求名字服务器处理递归查询。如果该位为 0，且被请求的名字服务器没有一个权威回答，则进行迭代查询，服务器将返回一个能解答该查询的其他名字服务器列表。
- RA：递归可用(Recursion Available)，如果名字服务器支持递归查询，则在响应中该比特置为 1。
- Z：必须为 0，保留字段。
- RCODE: 响应码(Response coded)，用于响应报，值为 0 则没有差错，值为 3 表示指定域名不存在。

- QDCOUNT：Question 域中的问题条数。在实际应用中，**QDCOUNT=1**。
- ANCOUNT：Answer 域中的 RR 条数。
- NSCOUNT：Authority 域中的 RR 条数。
- ARCOUNT：Additional 域中的 RR 条数。

1.2 Question

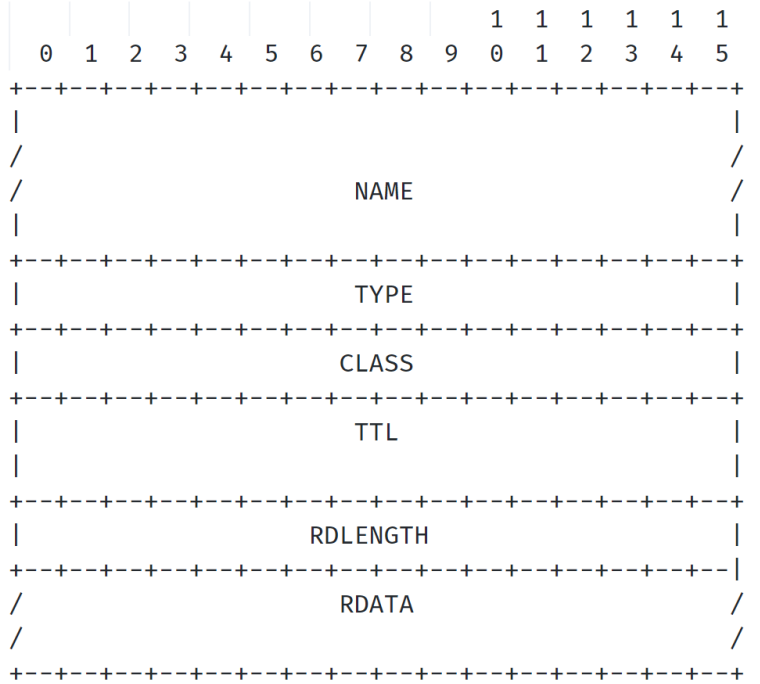
Question 域中的每个条目有如下格式：



- QNAME：要查询的域名，以 0 结尾，域名格式类似于数据链路层中的字符计数方法。
- QTYPE：查询的数据类型，用于区分 A(1)、MX(15)、CNAME(5)等。
- QCLASS：查询方式的类型，比如 IN(1)代表 Internet。

1.3 Resource Record

Answer/Authority/Additional 中的条目均为 Resource Record 格式：



- NAME：此条数据对应的域名。

- TYPE：RR 的类型码，A、AAAA、MX 等。
- CLASS: 通常为 IN(1)，指 Internet 数据。
- TTL：客户程序保留该资源记录的秒数。
- RDLENGTH：资源数据长度。
- RDATA：资源数据。

2 功能设计

2.1 基本功能

设计一个 DNS 服务器程序，读入“域名-IP 地址”对照表 `dnsrelay.txt`，当客户端查询域名对应的 IP 地址时，用域名检索该对照表，有三种可能：

1. 检索结果为 ip 地址 0.0.0.0，为不良网站，向客户端返回“域名不存在”的报错消息。
2. 检索结果为普通 IP 地址，为正常网站，向客户返回这个地址。
3. 表中未检到该域名，则缓存询问的来源并且向因特网 DNS 服务器发出查询，在收到上游 DNS 服务器的答案时将答案发给询问的来源客户端。

2.2 缓存

为了提高中继服务的效率，在本地开一个缓存区来存储近期向上游服务器查询得到的答案。缓存区的数据按照 LRU 的策略进行替换，并且要将 TTL 衰减到 0 的数据及时删除。

2.3 实现方法

DNS 中的查询报文本质上是对（域名，类型）数据的双关键字查询。为了方便处理，我将**表示类型的两个字节的数据添加到域名字符串的前面**，用得到的串作为哈希表中的关键字。在哈希表中，存储在这组询问中应该返回的完整报文（dnsrelay.txt 中的域名需要自己生成 A 记录的报文，其他类型需要按顺序存储从上游服务器发来的所有 resource record 项）。

缓存区的替换策略采用 LRU。维护一个新的链表，对于缓存区的每个数据，都在其中维护一个节点。当缓存区的一条数据被访问时，将其对应的链表节点移动到链表首部。当缓存区满了之后，删掉链表尾部的节点对应的数据。

为了及时清除 TTL 超限的数据，还需要使用一个堆，按照每个数据的过期时间进行排序。每次操作之前，都弹出堆顶直至堆顶的数据尚未过期。由于堆还需要支持由 LRU 的删除策略带来的内部删除，我选用了左偏树实现。

3 模块划分

3.1 HexString

对于代码中需要广泛用到的字节串，建立结构体

```
typedef struct __HEX_STRING {
```

```
    const uint8_t *s;
    int len;
} HexString;
```

实现了下列函数：

```
uint64_t HexStringHash(HexString s) {
    uint64_t h = 0;
    for (int i = 0; i < s.len; ++i) h = h * 257 + s.s[i];
    return h;
}

HexString HexStringCopy(HexString s) {
    uint8_t *p = malloc(s.len);
    memcpy(p, s.s, s.len);
    return (HexString){p, s.len};
}

int HexStringEqual(HexString a, HexString b) {
    return a.len == b.len && memcmp(a.s, b.s, a.len) == 0;
}
```

3.2 哈希表

哈希表用于对于给定的查询格式域名进行映射，使用挂链表的方式解决冲突。

哈希表节点定义如下：

```
struct __HASH_NODE {
    HashNode *next;
    uint64_t h;
    HexString s, v;
    ListNode *lisp;
    HeapNode *hep;
};
```

其中 lisp,hep 分别表示这条数据对应的链表节点、堆节点的指针。实现了如下函数：

```
void HashInsert(HexString s, HexString v, ListNode *lisp, HeapNode
*hep) {
    uint64_t h = HexStringHash(s);
    **--freeHashNode =
        (HashNode){hashList[h % HASH_MODULE], h, s, v, lisp, hep};
    hashList[h % HASH_MODULE] = *freeHashNode;
}

void HashDelete(HexString s, ListNode **lisp, HeapNode **hep) {
    uint64_t h = HexStringHash(s);
    for (HashNode *c = hashList[h % HASH_MODULE], *p = NULL; c;
        p = c, c = c->next)
```

```
            if (c->h == h && HexStringEqual(c->s, s)) {
                if (!p)
                    hashList[h % HASH_MODULE] = c->next;
                else
                    p->next = c->next;  // fix
                *freeHashNode++ = c;
                free((void *)c->s.s);
                free((void *)c->v.s);
                *lisp = c->lisp;
                *hep = c->hep;
                return;
            }
        assert(0);
    }

    HexString HashQuery(HexString s, ListNode **lisp) {
        uint64_t h = HexStringHash(s);
        for (HashNode *c = hashList[h % HASH_MODULE]; c; c = c->next)
            if (c->h == h && HexStringEqual(c->s, s)) {
                *lisp = c->lisp;
                return c->v;
            }
        return (HexString){NULL, -1};
    }
```

3.3 堆

堆用于按照过期时间将数据排序。堆节点定义如下：

```
    struct __HEAP_NODE {
        Heap
        Node *son[2], *fa;
        int dis;
        uint32_t expr;
        HashNode *hasp;
    };
```

实现了如下函数：

```
    HeapNode *HeapMerge(HeapNode *a, HeapNode *b) {
        if (!a || !b)
            return a ? a : b;
        else {
            if (a->expr > b->expr) Swap(a, b);
            a->son[1] = HeapMerge(a->son[1], b);
            a->son[1]->fa = a;
```

```
        if ((a->son[0] ? a->son[0]->dis : 0) < (a->son[1] ?
a->son[1]->dis : 0))
            Swap(a->son[0], a->son[1]);
        a->dis = a->son[1] ? a->son[1]->dis + 1 : 0;
        return a;
    }
}

void HeapPush(uint32_t expr, HashNode *hasp) {
    HeapNode *n = *--freeHeapNode;
    *n = (HeapNode){{NULL, NULL}, NULL, 0, expr, hasp};

    heapRoot = HeapMerge(heapRoot, n);
}

void HeapPop(HeapNode *p) {
    if (p == heapRoot)
        heapRoot = HeapMerge(p->son[0], p->son[1]);
    else {
        int d = p == p->fa->son[1];
        if ((p->fa->son[d] = HeapMerge(p->son[0], p->son[1])))
            p->fa->son[d]->fa = p->fa;
    }
    *freeHeapNode++ = p;
}
```

3.4 链表

链表用于配合哈希表进行 LRU 的维护。链表节点定义如下：

```
struct __LIST_NODE {
    ListNode *prev, *next;
    HexString s;
};
```

实现了如下操作：

```
void ListInsert(ListNode *n) {
    if (!listHead)
        listHead = listTail = n;
    else {
        listHead->prev = n;
        n->next = listHead;
        listHead = n;
    }
}
void ListExtract(ListNode *p) {
```

```
    if (listHead == listTail)
        listHead = listTail = NULL;
    else if (p == listHead) {
        listHead = p->next;
        p->next->prev = NULL;
    } else if (p == listTail) {
        listTail = p->prev;
        p->prev->next = NULL;
    } else {
        p->prev->next = p->next;
        p->next->prev = p->prev;
    }
    p->prev = p->next = NULL;
}
```

3.5 Cache

定义了一系列 cache 的操作：

```
void CacheInsert(HexString s, HexString v, uint32_t expr) {
    s = HexStringCopy(s);  // all from here
    v = HexStringCopy(v);
    **--freeListNode = (ListNode){NULL, NULL, s};
    ListInsert(*freeListNode);
    HeapPush(expr, NULL);
    HashInsert(s, v, *freeListNode, *freeHeapNode);
    (*freeHeapNode)->hasp = *freeHashNode;
}

void CacheRemove(HexString s) {
    // DEBUG("Remove ");
    PutHex(s.s, s.len);
    ListNode *lisp;
    HeapNode *hep;
    HashDelete(s, &lisp, &hep);
    ListExtract(lisp);
    *freeListNode++ = lisp;
    HeapPop(hep);
}

void CacheDiscard(int force) {
    uint32_t curTime = TimeStamp();
    while (heapRoot && heapRoot->expr <= curTime)
        CacheRemove(heapRoot->hasp->s);
    if     (force     &&     freeListNode     ==     freeListNodes)
```

```
CacheRemove(listTail->s);
}
```

3.6 DNS 报文操作

将 rc 个 resource record 报文的 TTL 缩小 d：

```
void DecreaseTTL(uint8_t *p, int rc, uint32_t d) {
    while (rc--) {
        p += p[0] & 0xc0 ? 2 : strlen((char *)p) + 1;
        DNSRecord *r = (DNSRecord *)p;
        DEBUG("ttl=%d d=%d\n", ntohl(r->ttl), d);
        if (ntohl(r->ttl) > d)
            r->ttl = htonl(ntohl(r->ttl) - d);
        else {
            DEBUG("shouldn't be!!!\n");
            r->ttl = htonl(0);
        }
        p += DNS_RECORD_SIZE + ntohs(r->rdlen);
    }
}
```

在 buffer 处使用 rs 中存储的内容建立一个完整的包：

```
int GenResponse(uint8_t *buffer, HexString rs) {
    // only modifies answer
    DNSHeader *header = (DNSHeader *)buffer;
    // header->id stay still
    uint16_t mask = ntohs(header->mask);
    mask &= ~((1 << 4) - 1);  // reply code 0
    mask &= ~(1 << 7);        // recursion unavailable
    mask &= ~(1 << 10);       // not authoritive
    mask |= (1 << 15);        // response message
    int len = DNS_HEADER_SIZE;
    buffer += DNS_HEADER_SIZE;
    len += strlen((char *)buffer) + 1 + DNS_QUESTION_SIZE;
    buffer += len - DNS_HEADER_SIZE;
    if (!rs.s) {
        mask |= 3;
        header->ancount = header->nscount = header->arcount = 0;
    } else {
        AnsHeader *ah = (AnsHeader *)rs.s;
        header->ancount = htons(ah->ancount);
        header->nscount = htons(ah->nscount);
        header->arcount = htons(ah->arcount);
        len += rs.len - ANS_HEADER_SIZE;
        memcpy(buffer,   rs.s   +   ANS_HEADER_SIZE,   rs.len   -
```

```
ANS_HEADER_SIZE);
        DecreaseTTL(buffer, ah->ancount + ah->nscount + ah->arcount,
                    TimeStamp() - ah->arrive);
    }
    header->mask = htons(mask);
    return len;
}
```

下列操作实现了将一个完整的DNS包的读取和解包操作。对于一个询问返回多组resource record的情况，整个包的TTL为所有项目TTL的最小值。

```
void ReadRecord(uint8_t *p, int *len, uint32_t *ttl) {
    *len = p[0] & 0xc0 ? 2 : strlen((char *)p) + 1;
    p += *len;
    *ttl = ntohl(((DNSRecord *)p)->ttl);
    *len += DNS_RECORD_SIZE + ntohs(((DNSRecord *)p)->rdlen);
}

int ReadRecords(uint8_t *buffer, int rc, uint32_t *ttl) {
    *ttl = -1;
    int len = 0;
    while (rc--) {
        int clen;
        uint32_t cttl;
        ReadRecord(buffer, &clen, &cttl);
        *ttl = Min(*ttl, cttl);
        buffer += clen;
        len += clen;
    }
    return len;
}

uint32_t Depacket(uint8_t *buffer, uint32_t tstamp, uint16_t
ancount,
                uint16_t nscount, uint16_t arcount) {
    memset(qsBuffer, 0, sizeof(qsBuffer));
    int l = strlen((char *)buffer);
    *((uint16_t *)qsBuffer) = *((uint16_t *)(buffer + l + 1));
    memcpy(qsBuffer + 2, buffer, l);
    qslen = l + 2;
    buffer += l + 1 + DNS_QUESTION_SIZE;
    memset(asBuffer, 0, sizeof(asBuffer));
    uint32_t ttl;
    aslen = ReadRecords(buffer, ancount + nscount + arcount, &ttl);
    AnsHeader *ah = (AnsHeader *)asBuffer;
    ah->arrive = tstamp;
```

```
    ah->ancount = ancount;
    ah->nscount = nscount;
    ah->arcount = arcount;
    memcpy(asBuffer + ANS_HEADER_SIZE, buffer, aslen);
    aslen += ANS_HEADER_SIZE;
    return ttl;
}
```

3.7 Initialize

将 dnsrelay.txt 中的条目读入，生成对应的 A 记录报文，并且加入 cache。
初始化本机的 socket 对象。

```
void Initialize() {
    logFile = fopen("prog.log", "w");
    for (int i = 0; i < CACHE_CAP; ++i) {
        *freeListNode++ = listNodePool + i;
        *freeHeapNode++ = heapNodePool + i;
    }
    for (int i = 0; i < HASH_CAP; ++i) *freeHashNode++ = hashNodePool
+ i;
    FILE *fp = fopen("dnsrelay.txt", "r");
    for (char ip[1024], dom[1024]; ~fscanf(fp, "%s %s", ip, dom);)
{
        ConvertDomain(dom);
        HexString hdom =
            HexStringCopy((HexString){(uint8_t *)dom, strlen(dom)});
        if (strcmp(ip, "0.0.0.0") == 0)
            HashInsert(hdom, (HexString){NULL, 0}, NULL, NULL);
        else {
            int alen = GenAnswer(ip);
            HexString hip = HexStringCopy((HexString){(uint8_t *)ip,
alen});
            HashInsert(hdom, hip, NULL, NULL);
        }
    }
    fclose(fp);
    for (int i = 0; i < (1 << ID_BITS); ++i) *freeQueID++ = i;
#ifndef linux
    WSADATA wsa;
    assert(WSAStartup(MAKEWORD(2, 2), &wsa) == 0);
#endif
    struct sockaddr_in localAddr;
    localSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    localAddr.sin_family = AF_INET;
```

```
    localAddr.sin_addr.s_addr = inet_addr("0.0.0.0");
    localAddr.sin_port = htons(53);
#ifndef linux
    u_long mode = 1;
    ioctlsocket(localSocket, FIONBIO, &mode);
#else
    fcntl(localSocket, F_SETFL, O_NONBLOCK);
#endif

    if       (bind(localSocket,       (SOCKADDR       *)&localAddr,
sizeof(SOCKADDR)) != 0) {
#ifdef linux
        fprintf(stderr, "errno is: %s\n", strerror(errno));
#endif
    }
    DNSAddr.sin_family = AF_INET;
    DNSAddr.sin_addr.s_addr = inet_addr(DNS_SERVER);
    DNSAddr.sin_port = htons(53);
}
```

3.8 Round

监听 53 端口，处理收到的报文。如果来自上层 DNS 服务器，则缓存发来的报文，并且将数据包发回给发起此次询问的客户；如果来自客户端，则在 cache 中查询是否有询问的答案，如果有则直接回发，否则记录数据包 ID 和客户的地址，向上层 DNS 服务器询问。

每次处理报文之前，都查找一下 cache 中已经过期的数据并且删除。

```
void Round() {
    struct sockaddr_in senderAddr;
    int senderAddrSize = sizeof(senderAddr);
    int len = recvfrom(localSocket, (char *)buffer, sizeof(buffer),
0,
                       (SOCKADDR *)&senderAddr, &senderAddrSize);
    if (len != -1) {
        DEBUG("%u %d\n", TimeStamp(), len);
        for (int i = 0; i < len; ++i) DEBUG("%02hx ", buffer[i]);
        DEBUG("\n");

        DNSHeader *h = (DNSHeader *)buffer;
        if (h->qdcount == htons(0x1)) {
            uint16_t real = ntohs(h->id);
            uint32_t tstamp = TimeStamp();
            uint32_t   ttl   =   Depacket(buffer   +   12,   tstamp,
ntohs(h->ancount),
                                  ntohs(h->nscount),
```

```
ntohs(h->arcount));
          HexString qs = (HexString){qsBuffer, qslen},
                    as = (HexString){asBuffer, aslen};
          if (senderAddr.sin_addr.s_addr ==
             DNSAddr.sin_addr.s_addr) {   // response from upper
server
             uint16_t cid = clientID[real];
             h->id = htons(cid);

             DEBUG("response from upper DNS server\n");
             for (int i = 0; i < len; ++i) DEBUG("%02hx ", buffer[i]);
             DEBUG("\n");

             sendto(localSocket, (char *)buffer, len, 0,
                   (SOCKADDR                 *)&clientAddr[real],
sizeof(clientAddr[real]));
             *freeQueID++ = real;
             CacheDiscard(1);
             assert(freeListNode != freeListNodes);
             CacheInsert(qs, as, tstamp + ttl);
          } else {  // query from lower server
             CacheDiscard(0);
             ListNode *lisp;
             HexString rs = HashQuery(qs, &lisp);
             if (rs.len >= 0) {
                int blen = GenResponse(buffer, rs);
                ListExtract(lisp);
                ListInsert(lisp);

                DEBUG("send cached answer\n");
                for (int i = 0; i < blen; ++i) DEBUG("%02hx ",
buffer[i]);
                DEBUG("\n");

                sendto(localSocket, (char *)buffer, blen, 0,
                      (SOCKADDR                   *)&senderAddr,
sizeof(senderAddr));
             } else {
                if (freeQueID != freeQueIDs) {
                   uint16_t idtoUpper = *--freeQueID;
                   clientAddr[idtoUpper] = senderAddr;
                   clientID[idtoUpper] = real;
                   h->id = htons(idtoUpper);
```

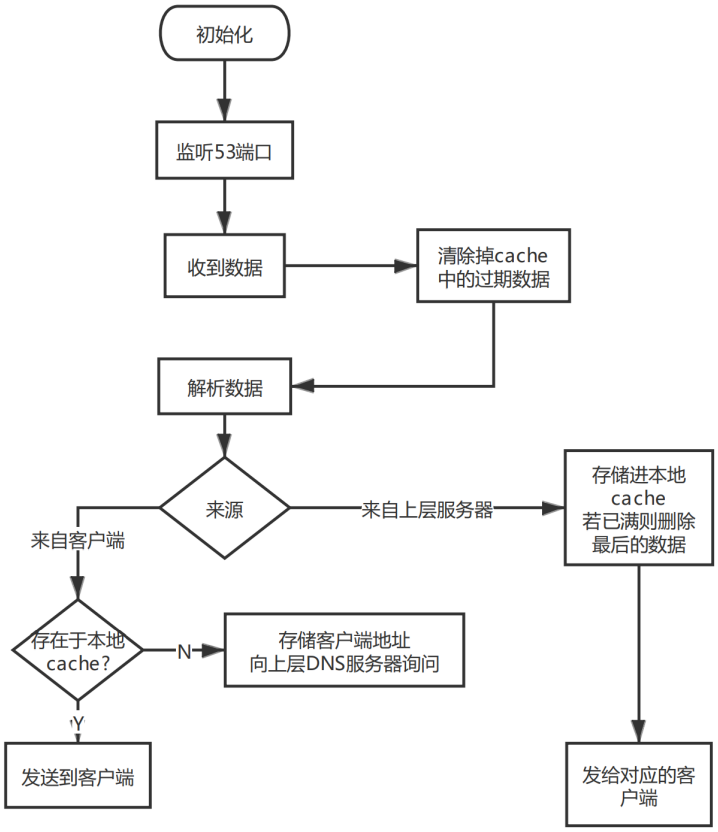
```
                    DEBUG("send query to upper DNS server\n");
                    for (int i = 0; i < len; ++i)
                        DEBUG("%02hx ", buffer[i]);
                    DEBUG("\n");

                    sendto(localSocket, (char *)buffer, len, 0,
                          (SOCKADDR *)&DNSAddr, sizeof(DNSAddr));
                } else
                    DEBUG("%u full client poll\n", TimeStamp());
            }
        }
    } else
        DEBUG("#    trash    on    port    53,    from    %u\n",
senderAddr.sin_addr.s_addr);
    }
}
```

3.9 主循环

初始化之后，循环调用 Round()函数，检测有没有收到报文。

```
for (Initialize();; Round());
```

4 软件流程图



5 测试用例以及运行结果

将服务部署在服务器 47.98.32.107，使用命令：

```
PS C:\Users\Lucida> nslookup github.com 47.98.32.107
```

服务器： UnKnown

Address： 47.98.32.107

非权威应答：

名称： github.com

Address： 13.250.177.223

将本地 DNS 服务器改为上述 IP 地址，运行数日之后网络正常，可以看出服务实现问题不大。

6 调试中遇到并解决的问题

C 中没有泛型编程，因此同一个函数会对不同字节数的变量有着不同的实现。我将一个 32 位整

数传入了 ntohs，编译器并未报警告，在运行过程中直接将传入的整数截断，使得程序运行出错。

struct 为了提高效率使用了在尾部填充的策略来做到内存对齐，因此 sizeof 得到的值并不是所有变量的大小之和。在处理包的过程中，我一开始使用了 sizeof 获得结构体的大小来进行指针的偏移，结果跳过了一些字节，出现了比较严重的问题。

我还曾经出现了数据结构中出现垃圾值的问题，后来发现是缓存区数组开小了。

7 心得体会

运行在实际环境中的程序比较复杂，在动手之前需要良好的设计。这次的堆+链表+哈希的数据结构组合是我在编程的过程中逐渐完善得到的，在数据结构之间的耦合的工作中浪费了一定的时间。在以后的学习和工作中，我会对复杂的体系充分设计再下手实现。